Creating RESTful Web Services with Grails

Eric Foster-Johnson
Principal Consultant, ObjectPartners

Using REST for APIs

REST helps you create simple lightweight APIs over HTTP using HTTP methods (verbs):

- GET - query data by ID
  - Can also search
- DELETE - remove object/row by ID
- POST - create new object/row
- PUT - update object/row
  - Lots of arguments over PUT vs POST

REST and Grails

- What is REST?
- REST for APIs
- RESTful purity
- What is Grails?
- Grails support for REST
- Highlights of Grails 3.0
- Finding out more

REST Resources

- A **resource** is an object in the system, such as a row in a database
- A **resource identifier** is what you use to find information about that resource
- Resource identifiers are usually URIs
  - company/2
  - employee/55
  - company/2/division/1

What is REST?

- Representational State Transfer (ReST)
- Architectural style and constraints for client-server communication
  - Client-server, Stateless, cacheable
  - Uniform interface, layered, code on demand (opt)
- Usually sent over HTTP
- Competes with SOAP, CORBA, RPC
- Defined by Roy Fielding in 2000
  - Paper is dense with abstract concepts

GET Requests

A GET request returns a representation of the resource (object or row) usually in JSON or XML

company/5

```
{"id":5,"name":"Cyberdyne Systems Corporation"}
```

```
<company id="5">
  <name>Cyberdyne Systems Corporation</name>
</company>
```

Inserting data

A POST request typically inserts a new row:

curl -i -X POST -H "Content-Type: application/json" -d '{"name":"Yoyodyne Propulsion Systems"}' localhost:8080/rest1/company

HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
Location: http://localhost:8080/rest1/company/7
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Mon, 23 Feb 2015 13:27:24 GMT

{"id":7,"name":"Yoyodyne Propulsion Systems"}

Why use REST?

- Relatively easy to create APIs
- Relatively easy to use APIs
- Can use your preferred programming language

RESTful purity

- HATEOAS - Hypermedia as the Engine of Application State - is a REST constraint
- In true REST you expose one URL
- The response contains links for supported operations
- Like hypertext (think company home pages)
- Downside: you usually need a human to interpret the results
- Useful for pagination (next and previous)

Why use Grails for REST?

- Grails has extensive REST support
  - If you can use defaults, REST is really easy
- Grails reduces software development time
- Grails encourages and supports testing

Real-life REST usage

- REST is used by many organizations to expose APIs
  - Twitter
  - LinkedIn
  - Salesforce.com
  - JIRA
  - and many more
- Many are not true REST
  - Don't worry too much about it

What is Grails?

- A Java EE enterprise application framework
- Builds to WAR files, deploys to Java EE servers, or executable JARs
- Coded in Java and Groovy, a JVM-based language
- Currently at version 2.4.4, with 3.0 coming in March 2015
- Grails 1.0 came out in February 2008
- Proven to be 30%-45% more productive in real projects
- Used by many companies such as:
  - Netflix
  - Virtuwell
  - United Health
  - Nationwide Insurance
  - Target
  - and many more...

Grails technology stack

Proven technology stack, including:

- Spring framework
  - Including transaction support
  - Next major version (3.0) will use Spring Boot
- Spring MVC (Web tier)
- Hibernate
- Groovy language
- JUnit and Spock for unit tests
- Built-in Tomcat for testing Web applications
- Loggers injected into services, controllers, etc.
- H2 in-memory database (for testing)

Extensive usage and documentation

- Minnesota is quite a hub for Grails usage
  - Used in Virtuwell, Target, Best Buy, MDE, MDH, etc.
  - Groovy Users of MN user group
  - GR8 US Conference
- Extensive documentation
  - http://grails.org/doc/latest/guide/
  - http://groovy.codehaus.org/
- Books, Blogs, Articles abound
  - http://grails.org/Books

Grails reduces development time

- Conventions make it quicker for new developers to come up to speed
  - Services, controllers are located in the same place in each project
- Defined build system and project layout
- Known location for configurations
- Known location for dependency definitions
- Active-record pattern simplifies database access
- No need to write boiler-plate code
- Groovy is much more productive than Java
- Extensive support for Unit Testing
- Note some changes coming in Grails 3.0

Groovy

- Grails makes heavy use of Groovy
- JVM-based language similar to Java, but less verbose
- Supports dynamic typing
  - Also static typing if desired
- Syntax support for maps, lists, regexes, closures
- Extends JDK, read in a file with one line of code, etc.
- Generates getters and setters
- Integrates well with Java, can run Java code, access Java classes
- Use *.groovy* file-name extension
- Compiled to .class files, just like Java

Convention over configuration

- Grails auto-wires code based on simple naming conventions
  - Spring auto-wiring
- Classes in *grails-app/domain* folder map to DB tables
  - Automatically supports IDs and version numbers
- Classes in *grails-app/services* folder are transactional services
- Controller class names end with *Controller*
- Bootstrap classes initialize data
  - Especially useful for testing

Groovy lists

```
// Lists
def list = ['A','B']

list << 5            // add to list
```

Groovy maps

```
// Maps
def map = [foo:'bar']

map.foo = 'bar'
```

Groovy conveniences - equals

```
if (name == "Bob") {   // Calls .equals() for you

}
```

Groovy conveniences - file

```
def listOfLines = new File('foo.txt').readLines()
```

Groovy conveniences - null safe

```
// Null-safe operator
if (user?.address?.street) {

}
```

Groovy conveniences - dates

```
def today = new Date()

Date tomorrow = today + 1
```

Groovy conveniences - Elvis

```
// Elvis operator

def displayName = user.name ?: "Anonymous"
```

Groovy conveniences - JSON

```
String result = new JsonBuilder(data:dataMap).toPrettyString()
```

GORM

- Object-relational mapper for Grails
- Based on Hibernate
  - Supports most relational DBs (Oracle, Sql Server, MySql, etc.)
  - Supports many NoSQL DBs (MongoDB, etc.)
- Easy transaction support
- Maps database tables to domain classes
- Maps table columns to domain class properties
- Conventions make it simpler than using Hibernate or JPA directly
- Hibernate creates schema in DB from domain classes (if asked to do so)
- In-memory DB (H2) for local development
- Built in dbconsole to view database (in development environment)
  - Super-handy feature
  - http://localhost:8080/*appName*/dbconsole

More Groovy conveniences

- "Insert ${variable} into string"
- Map-based constructor:

```
Person p = new Person(firstName:'Bob', lastName:'Johnson')
```

- Collect name from each item in collection:

```
def names = people*.name
```

Active record pattern

- Puts persistence support into domain classes
- No need for DAO layer:

```
Person person = new Person(firstName:'Bob',
   lastName:'Johnson')
person.save()   // Should wrap in transaction!

def employees = Person.list(offset:10, max:20,
       sort:"lastName", order:"asc")
```

Groovy closures

```
// Closures
assert [1, 2, 3].collect { it * 2 } == [2, 4, 6]

assert [1, 2, 3].find { it > 1 } == 2

assert [1, 2, 3].findAll { it > 1 } == [2, 3]

[1, 2, 3].each {
   println "Item: $it"
}
```

Dynamic finders

- Grails also creates dynamic finder methods
- Use *findBy** to find one
- Use *findAllBy** to find many

```
def minnesotans = Person.findAllByLastName('Johnson')

def recents = Person.findAllByDateCreatedBetween(
   firstDate, secondDate)
```

Domain classes

```
class Person {
    String firstName
    String lastName

    Date dateCreated
    Date lastUpdated

    static constraints = {
      firstName(blank: false, maxSize:30)
      lastName(blank: false, maxSize:40)
    }
}
```

Unit testing with Grails

Grails makes it easy to create tests

- **Unit** tests just run code under test
- **Integration** tests spool up the Grails environment
- **Functional** tests test a running Web application
  - Geb, Selenium, Spock functional, etc.

- Grails has JUnit 4 built in and well as **Spock** (in Grails 2.3+)
  - Can use geb, etc.
  - Supports many mocking frameworks

Relating domain classes

```
// One-to-many unidirectional
class Company {
    static hasMany = [ employees : Employee ]

    String name
}

// Make it bi-directional
class Employee {
    static belongsTo = [employer: Company]

    String firstName
}
```

JUnit test

```
@TestFor(Person)
class PersonTests {
    void testSomething() {
    String expectedResult = 'Marley'
    Person person = new Person(userName:'bobmarley',
            firstName: 'Bob',
            lastName: 'Marley', email:'bob@example.com')

    // Do something…

    assert person.lastName == expectedResult
    }
}
```

Extensive library of plugins

- A plugin holds Groovy/Java code and any other artifacts
  - Kind of like a library with CSS, images, JARs, etc.
  - Can write your own to modularize and share code
- Grails.org supports many add-ons
  - Spring Security
  - Fields plugin to customize Web UI
  - dbmigrations to manage schema changes
  - Build test data to create fake data
  - Many, many more
- See http://www.grails.org/plugins/

Spock test

```
class NumberSpec extends Specification {

 def "computing the maximum of two numbers"() {
    expect:
    Math.max(a, b) == c

    where:
    a << [5, 3]
    b << [1, 9]
    c << [5, 9]
 }
}
```

(From the Spock Basics wiki)

Grails support for REST

The nice things mentioned already make REST pretty easy with Grails. Grails also adds:

- Annotate a Domain class as a REST endpoint
- UrlMapping for a resource
- Create a controller that extends RestfulController
- Automatic detection of formats
  - XML, JSON
- Automatic conversion of objects to JSON, XML
  - Can customize if needed

Exposing domain classes

- Looks cool
- Trivial amount of code
  - Just the annotation - nothing else
- No one really uses this for real applications
  - Exposes database structure
  - You will likely need to customize a lot
- Probably need to create a controller

Expose domain class as endpoint

```
import grails.rest.Resource
@Resource(uri='/employees')        // That's it!!
class Employee {
   String firstName
   String lastName
}
```

Using UrlMappings

```
class UrlMappings {

  static mappings = {
     "/company/$id?"(resource: "company")
```

The resource specification maps HTTP operations to controller actions:

- GET - show()
- POST - save() // Save a new object
- PUT - update()
- DELETE - delete()

These are standard Grails actions for CRUD operations.

View domain endpoint data

- employees/1

```
<?xml version="1.0" encoding="UTF-8"?>
<employee id="1">
   <firstName>Bob</firstName>
   <lastName>Marley</lastName>
</employee>
```

- employees/1.**json**

```
{"class":"com.objectpartners.rest.Employee","id":1,"firstName":"Bob","lastName":"Marley"}
```

Manually creating controller

- You need to parse input and generate the XML or JSON response.
- withFormat helps

```
def show() {
  Person person = Person.get(params.id)
  withFormat {
     json {
        render person as JSON
     }
     xml {
        render person as XML
     },
     '*' {
        render "Error: only XML and JSON supported"
     }
  }
}
```

Creating RESTful controllers

- Manually creating REST controllers can be a lot of work
- Instead, you can extend RestfulController

```
import grails.rest.RestfulController
class CompanyController extends RestfulController {

   static responseFormats = ['json', 'xml']

   CompanyController() {
      super(Company)
   }
}
```

Customizing JSON rendering

- Grails offers a number of ways to customize JSON (or XML) rendering
- Here is the simplest:

```
JSON.registerObjectMarshaller(Company) {
  [
    id:it.id,
    name:it.name
  ]
}
```

Using RestfulController

RestfulController does a lot of work for you

- Does everything if you agree with its conventions
  - It is never that easy
- Still likely need to set up a resource in UrlMappings
  - To get rid of need for show, etc. in URLs
  - Add version numbers in the URL, /v1/person
- RestfulController can be customized
  - Override the methods you want to change
  - Can also customize JSON and XML rendering

HATEOAS with HAL and Atom

- Grails includes support for HATEOAS style links in the REST output.
- HalJsonRenderer outputs the HAL (Hypertext Application Language) format
  - _links
- AtomRenderer outputs the Atom format
  - <entry><link … >
- HATEOAS is useful for paginating results
  - Include links for next and previous

Customizing rendering

- Grails does a pretty good job of automatically rendering to JSON and XML
- It doesn't always do what you want
- JSON, for example, includes a class name, which exposes more than you probably want

```
{
  "class":"com.objectpartners.rest",
  "id":1,
  "name":"Weyland Corp"
}
```

Changes in Grails 3.0

- Based on Spring Boot
  - Builds runnable Jar with embedded Tomcat
- Uses Spring Boot configurations
  - Groovy, Yaml, XML, properties, classes
- Uses Boot dependency specifications
- Boot application class, with main()
- API changes with Traits
- Better IDE integration
- Builds based on Gradle
- Geb built in for functional tests

Finding out more

- One of the most useful sites is the REST cookbook at http://restcookbook.com/
- REST was defined in a paper by Roy T. Fielding. Read it online at http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- StackOverflow has an interesting discussion at http://stackoverflow.com/questions/671118/what-exactly-is-restful-programming
- Download Grails from https://grails.org/
- Read the Grails user guide at http://grails.org/doc/latest/
  - See the section on Web Services
- The ObjectPartners blog covers many Grails, Groovy and REST topics at http://www.objectpartners.com/blog/

Installing Grails

- Download Grails from grails.org
- You need a Java 7 JDK
- Unzip Grails
- Define GRAILS_HOME environment variable for root Grails directory.
- Add the Grails bin directory to the PATH environment variable
- If you use **gvm**, you can download and install with
  - gvm install grails 2.4.4